CAPÍTULO 7
ENCAPSULAMENTO

Encapsulamento é um conceito de Orientação a Objetos que define como os métodos e atributos de uma classe podem ser visualizados ou utilizados por outras classes. É a arte de esconder o que não é essencial, protegendo as informações e controlando o acesso aos dados e, portanto, impedindo o acesso indevido às informações do sistema. O conceito do encapsulamento pode ser identificado em diversos exemplos do cotidiano:

- **Celular**: para fazer uma ligação em um aparelho celular, basta digitar o número do telefone desejado e pressionar o botão que efetua ligação. Porém, diversos processos complexos são realizados pelo celular para que as pessoas possam conversar através dele. Se os usuários tivessem que ter conhecimento de todo o funcionamento interno do celular certamente a maioria destes não teria celular.

- **Carro**: a interface de uso de um carro é composta pelos dispositivos que permitem que o motorista conduza o veículo (volante, pedais, alavanca do cambio, etc). A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos. Para dirigir um carro só é necessário interagir com o volante, embreagem e pedais, mas não com o motor. Para trocar um carro a álcool para um a gasolina você não precisa reapreender a dirigir.

- **Máquina de refrigerantes**: Hoje em dia, é muito comum utilizar máquinas de refrigerantes, de salgadinhos, de doces, de café, etc. Normalmente, essas máquinas são extremamente protegidas, para garantir que nenhum usuário mal intencionado ou não tente alterar a implementação da máquina, ou seja, tente alterar como a máquina funciona por dentro.

7.1. MOTIVAÇÃO PARA ENCAPSULAR

Um dos problemas encontrados no nosso controle bancário é que o método SACA permite sacar mesmo que o limite tenha sido atingido. A seguir temos a codificação da classe CONTA:

```
public class Conta {

    public double saldo;
    public double limite;

    //...

    public void saca(double valor){
        this.saldo = this.saldo - valor;
    }

}
```

Podemos ultrapassar o limite da conta usando o método SACA:

```
public class TestaUltrapassaLimite {

    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = 2000.00;
        minhaConta.limite = 1000.00;
        minhaConta.saca(5000.00); // saldo + limite é só 3000!!
    }

}
```

Podemos incluir uma condição dentro do método SACA() para evitar que o valor de saque ultrapasse o valor do saldo mais limite, o que resultaria em uma conta em estado inconsistente, com seu saldo abaixo do limite:

```
public void saca(double valor){
    if (valor <= this.saldo + this.limite)
        this.saldo = this.saldo - valor;
}
```

Apesar disso, ainda temos um problema: ninguém garante que o usuário da classe utilizará o método para alterar o saldo da conta. O código a seguir ultrapassa o limite diretamente:

```
public class TestaUltrapassaLimite {

    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.limite = 1000.00;
        minhaConta.saldo = -2000.00;
    }

}
```

Uma forma simples de resolver esta situação seria realizar um teste para saber se não estamos ultrapassando o limite antes de alterarmos o saldo:

```
public class TestaUltrapassaLimite {

    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = 2000.00;
        minhaConta.limite = 1000.00;

        // quero mudar o saldo para -3000
        double novoSaldo = -3000;

        // testa se o novoSaldo ultrapassa o limite da conta
        if (novoSaldo < -minhaConta.limite)
            System.out.println("Não posso mudar para esse saldo");
        else
            minhaConta.saldo = novoSaldo;
    }

}
```

Esse código se repetiria ao longo de toda aplicação e alguém pode esquecer de fazer essa validação em algum momento, deixando a conta na situação inconsistente. A melhor forma de resolver isso seria forçar quem usa a classe Conta a invocar o método saca e não permitir o acesso direto ao atributo. É o mesmo caso da validação de CPF.

Considerando que todas as regras referentes a classe estão contidas na própria classe (e nunca em outra parte da aplicação), o acesso aos atributos deverão ser feitos de modo a garantir que tais regras sejam cumpridas.

7.2. CONTROLANDO O ACESSO

Para resolver o problema de acesso ao atributo saldo da classe CONTA, basta declararmos que os atributos não podem ser acessados de fora da classe através da palavra chave PRIVATE:

```
public class Conta {

    private double saldo;
    private double limite;

    //...

}
```

Quando definimos que um atributo (ou método) tem a visibilidade privada, limitamos o acesso somente à sua classe. Podemos imaginar que, depois de instanciado o objeto, esse atributo fica protegido (encapsulado) dentro desse objeto. Assim, não é possível acessá-lo de nenhuma outra classe ou objeto. Tal característica é extremamente importante para proteger atributos que não devem ter seus conteúdos vulneráveis a acessos indevidos. Marcando um atributo como privado, fechamos o acesso ao mesmo em relação a todas as outras classes, fazendo com que o seguinte código não compile:

```
public class TestaAcessoDireto {

    public static void main(String[] args) {
        Conta minhaConta = new Conta();

        //não compila! você não pode acessar o atributo privado de outra classe
        minhaConta.saldo = 2000.00;
        minhaConta.
    }
}
```

The field Conta.saldo is not visible
2 quick fixes available:
Change visibility of 'saldo' to 'default'
Create getter and setter for 'saldo'
Press 'F2' for focus

Na orientação a objetos, é prática quase que obrigatória proteger seus atributos com PRIVATE. Assim, cada classe será responsável por controlar seus atributos, julgando se um novo valor é válido ou não. Esta validação não deve ser controlada por quem está usando a classe e sim por ela mesma, centralizando essa responsabilidade e facilitando futuras mudanças no sistema.

Muitas outras vezes nem mesmo queremos que outras classes saibam da existência de determinado atributo, escondendo-o por completo, já que ele diz respeito ao funcionamento interno do objeto. Repare que, na invocação do método saca não fazemos a menor ideia de que existe um limite que está sendo verificado. Para quem for usar essa classe, basta saber o que o método faz e não como exatamente ele o faz.

Encapsular é fundamental para que os sistemas sejam suscetíveis a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está encapsulada. Imagine, por exemplo, cobrar CPMF de cada saque: basta modificarmos o método saca e nenhum outro código. Mais: as classes que usam esse método nem precisam saber desta modificação, bastando apenas recompilar a classe Conta e substituir aquele arquivo .class. Ganhamos muito em esconder o funcionamento do nosso método na hora de dar manutenção e fazer modificações.

7.3. ENCAPSULANDO MÉTODOS

A palavra chave PRIVATE também pode ser usada para modificar o acesso a um método. Tal funcionalidade é utilizada quando existe um método que serve apenas para auxiliar a própria classe e quando há código repetido dentro de dois métodos da classe.

```
public class PessoaFisica {

    private String nome;
    private String cpf;

    public void mudaCPF(String cpf) {
        validaCPF(cpf);
        this.cpf = cpf;
    }

    private void validaCPF(String cpf) {
        // série de regras aqui, falha caso nao seja válido
    }

    // ..
}
```

No exemplo da validação de CPF da classe PESSOAFISICA, o método VALIDACPF foi definido com o modificador private e não pode ser acessado fora da classe. Assim, o método que deverá ser invocado por quem usar esta classe será o MUDACPF. O controle sobre o CPF está centralizado: ninguém consegue acessá-lo sem passar pelo MUDACPF, a classe PESSOAFISICA é a única responsável pelos seus próprios atributos.

Sempre devemos expor o mínimo possível de funcionalidades, para criar um baixo acoplamento entre as nossas classes.

É muito comum, e faz todo sentido, que seus atributos sejam PRIVATE e quase todos seus métodos sejam public (não é uma regra!). Desta forma, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos.

7.4. MÉTODOS DE ACESSO

O modificador PRIVATE faz com que ninguém consiga modificar, nem mesmo ler, o atributo em questão. Com isso, temos um problema: como fazer para mostrar o saldo de uma Conta, já que nem mesmo podemos acessá-lo para leitura? Precisamos então arranjar uma forma de acessá-lo. Sempre que precisamos arrumar uma maneira de fazer alguma coisa com um objeto, utilizamos métodos. Vamos então criar um método, digamos LESALDO, para realizar essa simples tarefa:

```
public class Conta {

    private double saldo;
    //outros atributo omitidos

    public double leSaldo(){
        return this.saldo;
    }

    //deposita() e saca() omitidos
}
```

Para acessarmos o saldo de uma conta, podemos fazer:

```
public class TestaAcessoComExibeSaldo {

    public static void main(String[] args) {
        Conta minhaConta = new Conta();

        minhaConta.deposita(2500);

        System.out.println("Saldo: " + minhaConta.leSaldo());

    }

}
```

Para garantir o encapsulamento e possibilitar o acesso aos dados do objeto, são criados dois métodos de acesso aos atributos em casos específicos, um que retorna o valor e outro que muda o valor. Esses métodos são codificados dentro da classe e, por isso, definidos com acessibilidade pública (public). Significa que eles podem ser acessados a partir do objeto instanciado de qualquer classe. Servem, portanto, como portas de entrada e de saída de informações dos atributos, por onde seus valores podem ser lidos e alterados.

A vantagem de abrir esse acesso via método é poder definir (programar) as regras de leitura e escrita nesses atributos. Resumindo, nenhuma outra classe ou objeto pode acessar diretamente os atributos encapsulados. Se isso for necessário, deve-se "pedir" ao objeto para que ele mostre ou altere o valor de um de seus atributos mediante suas próprias regras (definidas nos métodos de acesso).

A convenção para esses métodos é de colocar a palavra get ou set antes do nome do atributo.

7.4.1. Método get

Por padrão do próprio Java, os métodos que leem e mostram (retornam) o conteúdo de um atributo começam com a palavra get, seguida pelo nome desse atributo.

```
public double getSaldo() {
    return saldo;
}
```

7.4.2. Método set

Já os métodos responsáveis por escrever (alterar) o conteúdo de um atributo começam com a palavra set, seguida pelo nome desse atributo.

```
public void setSaldo(double saldo) {
    this.saldo = saldo;
}
```

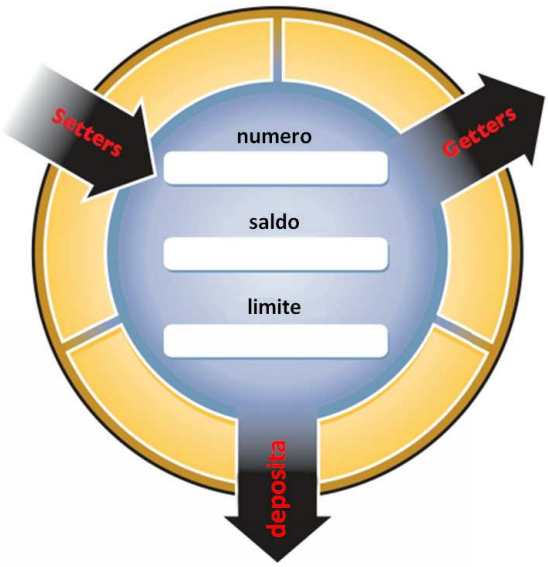
É uma má prática criar uma classe e, logo em seguida, criar getters e setters para todos seus atributos. Você só deve criar um getter ou setter se tiver a real necessidade.

Repare que nesse exemplo setSaldo não deveria ter sido criado, já que queremos que todos usem deposita() e saca().

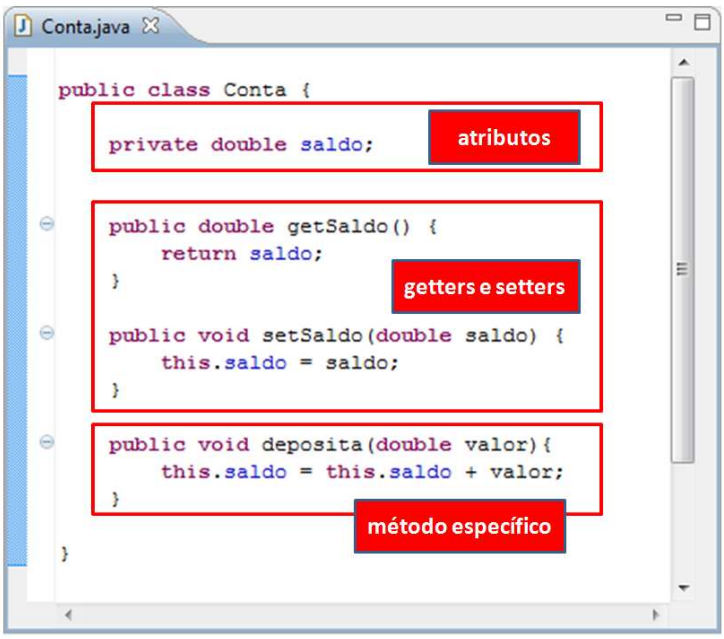
Utilizar getters e setters não só ajuda você a proteger seus atributos, como também possibilita ter de mudar algo em um só lugar... chamamos isso de encapsulamento, pois esconde a maneira como os objetos guardam seus dados. É uma prática muito importante.

7.5. VISÃO GERAL DA CLASSE CONTA

A figura seguinte ilustra o encapsulamento em um objeto do tipo Conta. Imagine o objeto como uma esfera, que possui na parte interna (azul) os atributos (privados) que não têm acesso ao mundo externo (restante da aplicação). Já a camada externa (amarela) tem acesso ao mundo externo (pelo lado de fora) e à parte interna (pelo lado de dentro) e funciona como uma película protetora do objeto. Tem passagens padrão que transferem informações de fora para dentro (setters) e de dentro para fora (getters). Além disso, têm algumas passagens (métodos) com finalidades específicas e que também podem entrar e/ou sair com informações. Esse tráfego acontece de acordo com regras de entrada e saída de informações definidas nas passagens (métodos). Em nossos exemplos, sempre serão utilizados GETTERS e SETTERS para acessar os atributos, garantindo os benefícios do encapsulamento.



A figura seguinte reproduz a codificação completa da classe Conta, destacando as camadas vistas até agora.



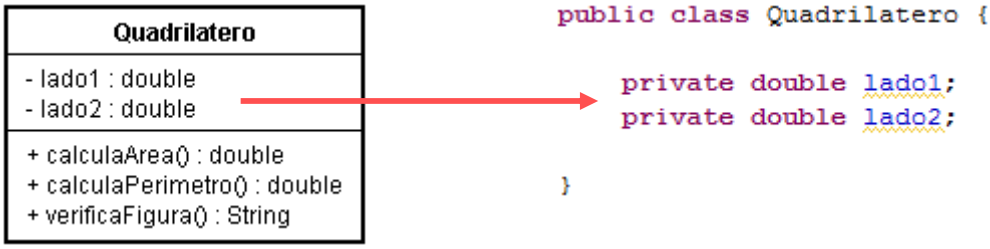
7.6. EXERCÍCIOS: Encapsulamento

1. Crie um novo projeto Java em **`File`** ➔ **`New`** ➔ **`Project`**.

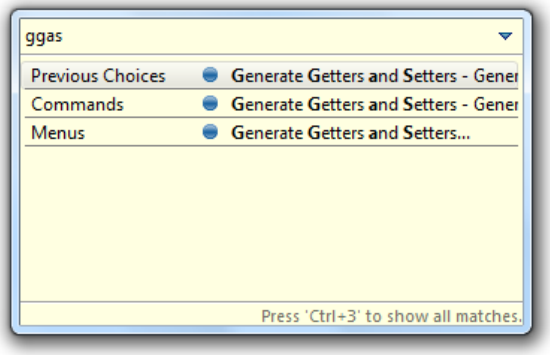
2. Selecione `Java Project` e clique em **Next**;

3. Coloque o nome do projeto como *`ds1-Encapsulamento`* e clique em **Finish**;

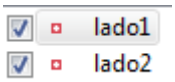
4. Crie a classe de modelagem a seguir e defina os atributos com o modificador `PRIVATE`:



5. Crie os métodos de acesso getters e setters:
   a) Aperte **Ctrl + 3**
   b) Digite *`ggas`* que é a abreviação de `Generate getters and setters`

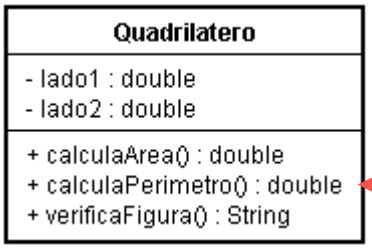


c) Pressione ENTER
d) Selecione todos os atributos.



e) Finalize com **Ok**
f) Organize o código com CTRL + SHIFT + F.

6. Codifique os métodos específicos da classe:




```
public double calculaArea(){
    return this.lado1 * this.lado2;
}

public double calculaPerimetro(){
    return (2*this.lado1) + (2*this.lado2);
}

public String verificaFigura(){
    if(this.lado1==this.lado2)
        return "É um quadrado";
    else
        return "É um retângulo";
}
```

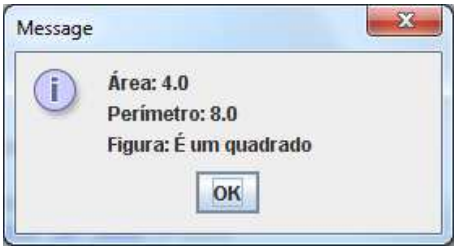
6) Crie a classe TestaQuadrilatero com o método *`main()`*.
   a. Instancie um objeto do tipo `Quadrilatero` chamado `retangulo`.

```
public static void main(String[] args) {

    Quadrilatero retangulo = new Quadrilatero();

}
```

   b. Instancie um objeto do tipo `DecimalFormat` chamado `df` e formate-o para uma casa decimal.
   c. Altere o atributos `LADO1` e `LADO2` do objeto `RETANGULO`. Não esqueça de usar o método de acesso `SET` para isso.

```
retangulo.setLado1(Double.parseDouble(JOptionPane
        .showInputDialog("Digite o valor do lado 1")));
```

   d. Exiba o resultado de todos os métodos do objeto `RETANGULO` com o método `SHOWMESSAGEDIALOG` da classe `JOPTIONPANE`. Os métodos deverão ter seus valores formatados com uma casa decimal.



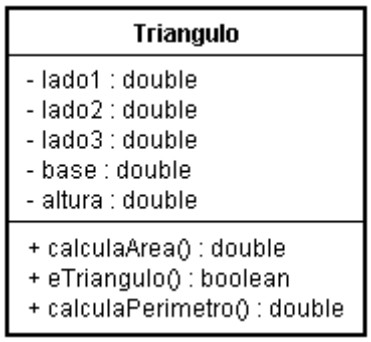
7.7. EXERCÍCIOS: Fixando conceitos

1. Crie um novo projeto Java em **`File`** ➔ **`New`** ➔ **`Project`**.

2. Selecione `Java Project` e clique em **Next**;

3. Coloque o nome do projeto como *`ds1-Triangulo`* e clique em **Finish**;

4. Crie a classe de modelagem a seguir (NÃO esqueça do *getters* e *setters*):



- **calculaArea**: faz o cálculo da área (base * altura)/2.
- **eTriangulo:** Retorna true se o comprimento de cada lado é menor que a soma dos outros dois lados. Senão for retorna false.
- **calculaPerimetro**: se o método eTriangulo() for igual a true, retorna o cálculo do perímetro. Caso contrário, retorne 0.0.

7) Crie a classe TestaTriangulo com o método *`main()`*.
   a. Instancie um objeto do tipo `Triangulo` chamado `tri`.
   b. Instancie um objeto do tipo `DecimalFormat` chamado `df` e formate-o para uma casa decimal.
   c. Altere os atributos `LADO1`, `LADO2`, `LADO3`, `BASE` e `ALTURA` do objeto `TRI`. Não se esqueça de usar o método de acesso `SET` para isso.
   d. Exiba o resultado de todos os métodos do objeto `TRI` com o método `SHOWMESSAGEDIALOG` da classe `JOPTIONPANE`. Os métodos deverão ter seus valores formatados com uma casa decimal.

7.8. CLASSE MATH

Fornece constantes e métodos estáticos para trigonométricas, logarítmicas e outras funções matemáticas comuns como, por exemplo:

Método	Descrição	Exemplo
Max(Valor, Valor)	Retorna o maior dos dois números.	`maior = Math.Max(valor1, valor2);`
Min(Valor, Valor)	Retorna o menor dos dois números.	`menor = Math.Max(valor1, valor2);`
Pi	Retorna o valor de π.	`pi = Math.PI;`
Pow	Retorna um número especificado elevado à potência especificada.	`potencia = Math.Pow(valor1, valor2);`

Sqrt	Retorna a raiz quadrada de um número especificado.	`raiz = Math.Sqrt(valor1);`

7.9. EXERCÍCIOS: Usando Math

1) Crie um novo projeto Java em **`File`** ➔ **`New`** ➔ **`Project`**.

2) Selecione `Java Project` e clique em **Next**;

3) Coloque o nome do projeto como *`ds1-classemath`* e clique em **Finish**;

4) Crie a classe a seguir com seus atributos e métodos:

Poco
- diametro : double - profundidade : double
+ raio () : double + volume () : double + validaPoco () : String

- **raio()**: Metade do diâmetro do poço.
- **volume():** Calcula a quantidade de água que um poço pode armazenar utilizando a equação: 2 * π * raio² * altura.
- **validaPoco ():** Verifica se o poço armazena pelo menos 5.000 litros de água.

5) Crie a classe TestaPoco com o método *`main()`*.
   a. Instancie um objeto do tipo `Poco` chamado `meuPoco`.
   b. Altere os atributos `DIAMETRO` e `PROFUNDIDADE` do objeto `MEUPOCO`. Não esqueça de usar o método de acesso `SET` para isso.
   c. Exiba o resultado do método validaPoco do objeto `MEUPOCO` com o método `SHOWMESSAGEDIALOG` da classe `JOPTIONPANE`.

Referência Bibliográfica:

Apostila Caelum - http://www.caelum.com.br/
http://www.globalcode.com.br/home
http://www.k19.com.br/downloads/apostilas-java
http://www.algaworks.com/
Barnes, David J. e Kolling, Michael. Programação Orientada a Objetos com Java. Pearson
Arnold, Ken, Gosling, James e Holmes, David. A Linguagem de Programação Java. Bookman
Harvey M. Deitel, Paul J. Deitel. Java Como Programar. 6ed. Pearson
Flanagan, David. Guia Essencial do Java. Bookman
Martin, Fowler. UML Essencial.
Livro de Informática – Centro Paula Souza